Example: Counting change

It takes only a bit of cleverness to come up with the iterative Fibonacci algorithm. In contrast, consider the following problem: How many different ways can we make change of $1.00, given half-dollars, quarters, dimes, nickels, and pennies? More generally, can we write a procedure to compute the number of ways to change any given amount of money?

This problem has a simple solution as a recursive procedure. Suppose we think of the types of coins available as arranged in some order. Then the following relation holds:

The number of ways to change amount a using n kinds of coins equals

- the number of ways to change amount a using all but the first kind of coin, plus

- the number of ways to change amount $a - d$ using all n kinds of coins, where d is the denomination of the first kind of coin.

To see why this is true, observe that the ways to make change can be divided into two groups: those that do not use any of the first kind of coin, and those that do. Therefore, the total number of ways to make change for some amount is equal to the number of ways to make change for the amount without using any of the first kind of coin, plus the number of ways to make change assuming that we do use the first kind of coin. But the latter number is equal to the number of ways to make change for the amount that remains after using a coin of the first kind.

Thus, we can recursively reduce the problem of changing a given amount to the problem of changing smaller amounts using fewer kinds of coins. Consider this reduction rule carefully, and convince yourself that we

can use it to describe an algorithm if we specify the following degenerate cases:[33]

- If *a* is exactly 0, we should count that as 1 way to make change.

- If *a* is less than 0, we should count that as 0 ways to make change.

- If *n* is 0, we should count that as 0 ways to make change.

We can easily translate this description into a recursive procedure:

```
(define (count-change amount)
  (cc amount 5))
(define (cc amount kinds-of-coins)
  (cond ((= amount 0) 1)
        ((or (< amount 0) (= kinds-of-coins 0)) 0)
        (else (+ (cc amount
                     (- kinds-of-coins 1))
                 (cc (- amount
                        (first-denomination kinds-of-coins))
                     kinds-of-coins)))))
(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 5)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 25)
        ((= kinds-of-coins 5) 50)))
```

(The `first-denomination` procedure takes as input the number of kinds of coins available and returns the denomination of the first kind. Here we

---

[33] For example, work through in detail how the reduction rule applies to the problem of making change for 10 cents using pennies and nickels.

are thinking of the coins as arranged in order from largest to smallest, but any order would do as well.) We can now answer our original question about changing a dollar:

```
(count-change 100)
292
```

Count-change generates a tree-recursive process with redundancies similar to those in our first implementation of fib. (It will take quite a while for that 292 to be computed.) On the other hand, it is not obvious how to design a better algorithm for computing the result, and we leave this problem as a challenge. The observation that a tree-recursive process may be highly inefficient but often easy to specify and understand has led people to propose that one could get the best of both worlds by designing a "smart compiler" that could transform tree-recursive procedures into more efficient procedures that compute the same result.[34]

> **Exercise 1.11:** A function f is defined by the rule that $f(n) = n$ if $n < 3$ and $f(n) = f(n-1) + 2f(n-2) + 3f(n-3)$ if $n \geq 3$. Write a procedure that computes f by means of a recursive process. Write a procedure that computes f by means of an iterative process.

---

[34] One approach to coping with redundant computations is to arrange matters so that we automatically construct a table of values as they are computed. Each time we are asked to apply the procedure to some argument, we first look to see if the value is already stored in the table, in which case we avoid performing the redundant computation. This strategy, known as *tabulation* or *memoization*, can be implemented in a straightforward way. Tabulation can sometimes be used to transform processes that require an exponential number of steps (such as count-change) into processes whose space and time requirements grow linearly with the input. See Exercise 3.27.